**React Hook**

**什么是Hook？**

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

Hook没有破坏性的改动，所以不会影响你对React概念的理解。

Hook是完全可选、100%向后兼容和现在可用的。

**Hook的动机是什么？**

解决编写和维护成千上万的组件时遇到的各种各样看起来不相关的问题，如在组建之间复用状态逻辑很难，复杂组件变得难以理解，以及难以理解的class。

Hook使用规则

Hook 就是 JavaScript 函数，使用它们会有额外的规则：

1.只能在函数最外层调用 Hook。不要在循环、条件判断或者子函数中调用。

2.只能在 React 的函数组件中调用 Hook。不要在其他 JavaScript 函数中调用。（还有一个地方可以调用 Hook —— 就是自定义的 Hook 中后面会介绍）

3.不能在 class 组件内部使用 Hook，但可以在组件树里混合使用 class 组件和使用了 Hook 的函数组件。

**使用State Hook**

**示例**

import React, { useState } from 'react';

function Example() {

// 声明一个叫 “count” 的 state 变量

const [count, setCount] = useState(0);

return

}

我们声明了一个叫 count 的 state 变量，然后把它设为 0。React 会在重复渲染时记住它当前的值，并且提供最新的值给我们的函数。我们可以通过调用 setCount 来更新当前的 count。

useState 会返回一对值：当前状态和一个让你更新它的函数，你可以在事件处理函数中或其他一些地方调用这个函数。它类似 class 组件的 this.setState，但是它不会把新的 state 和旧的 state 进行合并。

**值得注意的是：**

1.初始值不是必须的；

2.setState 不是一个同步的函数，在更新完state之后立即取值并不会得到最新的state

3. useState 主要用途是更新与组件相关的值，如果仅在js中使用的变量则不需要使用useState。

**使用Effect Hook**

**示例**

import React, { useState, useEffect } from 'react';

function Example() {

  const [count, setCount] = useState(0);

  useEffect(() => {

document.title = `You clicked ${count} times`;  });

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

通过使用这个 Hook，你可以告诉 React 组件需要在渲染后执行某些操作。React 会保存你传递的函数（我们将它称之为 “effect”），并且在执行 DOM 更新之后调用它。在这个 effect 中，我们设置了 document 的 title 属性，也可以执行数据获取或调用其他命令式的 API。

将 useEffect 放在组件内部让我们可以在 effect 中直接访问 count state 变量（或其他 props）。我们不需要特殊的 API 来读取它（比如this） 它已经保存在函数作用域中。

如果你熟悉 React class 的生命周期函数，你可以把 useEffect Hook 看做componentDidMount，componentDidUpdate 和 componentWillUnmount 这三个函数的组合。

如果你对  useEffect 仍然不是很理解 可以暂时简单的把其理解为

1.  useEffect  在同个组件内可多次调用 便于分离逻辑；

2. useEffect（（）=> {xxxxx}, [] ) 相当于componentDidMount 仅在第一次渲染后调用

3. useEffect（（）=> { xxxxx return NNNN } ） 匿名函数内 相当于componentDidUpdate 及componentDidMount 第一次渲染后和每次更新都会调用到。 return 部分相当于 componentWillUnmount 组件卸载前会调用。

4. useEffect( (）=> {}, [AA，BB] ) 由于每次渲染都会调用useEffect 所以如果要进行性能优化可以使用 函数的第二个可选参数，在AA或BB 更改时进行更新。如果你要使用此优化方式，请确保数组中包含了所有外部作用域中会随时间变化并且在 effect 中使用的变量

注：这只是个人的理解，如果需要更深刻的了解effect请仔细查询官方文档以便于正确的理解。

**值得注意的是**

1. 组件的销毁渲染在整个页面的周期当中可能会进行多次，不要依赖effect来进行某些判断

2. 传递给 useEffect 的函数在每次渲染中都会有所不同，这是刻意为之的。事实上这正是我们可以在 effect 中获取最新的 的值，而不用担心其过期的原因。每次我们重新渲染，都会生成新的 effect，替换掉之前的。某种意义上讲，effect 更像是渲染结果的一部分 —— 每个 effect “属于”一次特定的渲染。

3. 组件内如果有多个effect 按使用顺序依次执行。

**使用自定义Hook**

**示例**

export const useInitTable = (api:any, initFilters:any) => {

  const [filters, setFilters] = useState({

    pageIndex: 1,

    pageSize: 10,

    ...initFilters

  });

  // 表单数据

  const [list, setList] = useState();

  // 列表总数

  const [total, setTotal] = useState(0);

  // 是否在加载中

  const [loading, setLoading] = useState(false);

  useEffect(() => {

    getList({...filters});

    // eslint-disable-next-line

  }, [filters])

  // 获取列表

  const getList = async (filters:any) => {

    setLoading(true);

    const resData = await api(filters);

    setLoading(false);

    setList(resData.data.rows);

    setTotal(resData.data.total);

  }

  // 更改页数

  const changePage = (pagination: any) => {

    setFilters({

      ...filters,

      pageIndex: pagination.current,

      pageSize: pagination.pageSize

    })

  }

  return {

    filters,

    list,

    total,

    loading,

    changePage,

    setFilters,

    getList

  }

}

通过自定义 Hook，可以将组件逻辑提取到可重用的函数中。上述示例为table 提供了网络请求和分页的业务逻辑，通过filters的更改进行新请求以更新table的内容

**值得注意的是**

1.自定义 Hook 是一个函数，其名称以 “use” 开头，函数内部可以调用其他的 Hook。与 React 组件不同的是，自定义 Hook 不需要具有特殊的标识。我们可以自由的决定它的参数是什么，以及它应该返回什么（如果需要的话）。换句话说，它就像一个正常的函数。但是它的名字应该始终以 use 开头，这样可以一眼看出其符合 Hook 的规则。

2.自定义 Hook 必须以 “use” 开头吗？必须如此。这个约定非常重要。不遵循的话，由于无法判断某个函数是否包含对其内部 Hook 的调用，React 将无法自动检查你的 Hook 是否违反了 Hook 的规则。

3.在两个组件中使用相同的 Hook 会共享 state 吗？不会。自定义 Hook 是一种重用状态逻辑的机制(如上述示例储存当前table的网络请求和请求参数)，所以每次使用自定义 Hook 时，其中的所有 state 和副作用都是完全隔离的。

**使用Context Hook**

**示例**

const themes = {

  light: {

    foreground: "#000000",

    background: "#eeeeee"

  },

  dark: {

    foreground: "#ffffff",

    background: "#222222"

  }};

const ThemeContext = React.createContext(themes.light);

function App() {

  return

<ThemeContext.Provider value={themes.dark}>

<Toolbar />

</ThemeContext.Provider>

  );}

function Toolbar(props) {

  return (

<div>

<ThemedButton />

</div>

  );

}

function ThemedButton() {

 const theme = useContext(ThemeContext);

return (

<button style={{

background: theme.background,

color: theme.foreground }}>

 I am styled by theme context!

</button>

);}

接收一个 context 对象（React.createContext 的返回值）并返回该 context 的当前值。当前的 context 值由上层组件中距离当前组件最近的 <MyContext.Provider> 的 value prop 决定。

当组件上层最近的 <MyContext.Provider> 更新时，该 Hook 会触发重渲染，并使用最新传递给 MyContext provider 的 context value 值。即使祖先使用 React.memo 或 shouldComponentUpdate，也会在组件本身使用 useContext 时重新渲染。

Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言，如果对context不是很熟悉可以先了解context再看useContext

**值得注意的是**

1.调用了 useContext 的组件总会在 context 值变化时重新渲染。如果重渲染组件的开销较大，你可以 通过使用 memoization 来优化。

2.useContext(MyContext) 只是让你能够读取 context 的值以及订阅 context 的变化。你仍然需要在上层组件树中使用 <MyContext.Provider> 来为下层组件提供 context。

**以上介绍属于基础Hook，还有一些额外的Hook额外的 Hook：**

useReducer、useCallback、useMemo、useRef、useImperativeHandle、useLayoutEffect、useDebugValue 由于使用频率较低如有兴趣可前往react官网学习Hook API 索引 – React (docschina.org)